(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
10 May 2001 (10.05.2001)

PCT

(10) International Publication Number
WO 01/33352 A1

(51) International Patent Classification⁷: G06F 9/48, 9/38

(21) International Application Number: PCT/US00/20881

(22) International Filing Date: 31 July 2000 (31.07.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/431,358 1 November 1999 (01.11.1999) US

(71) Applicant: ADVANCED MICRO DEVICES, INC. [US/US]; One AMD Place, Mail Stop 68, Sunnyvale, CA 94088-3453 (US).

(72) Inventors: KRANICH, Uwe; Amalienweg 14, D-85551 Kirchheim (DE). CHRISTIE, David, S.; 6201 Needham Cove, Austin, TX 78739 (US).

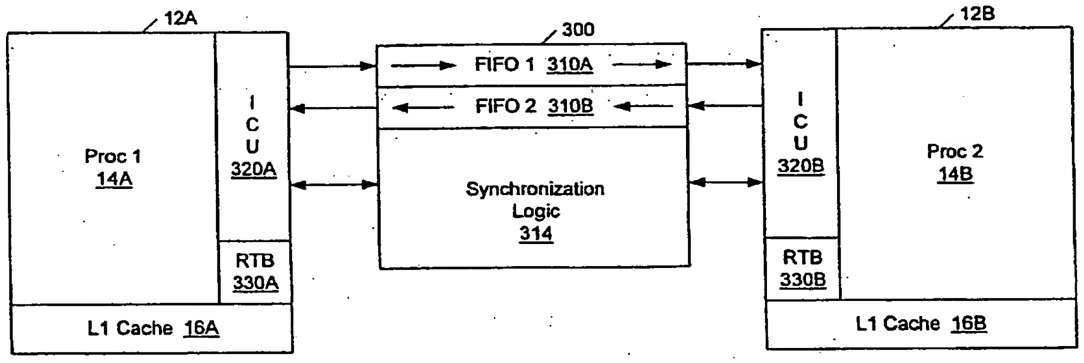
(74) Agent: DRAKE, Paul, S.; Advanced Micro Devices, Inc., Mail Stop 562, 5204 East Ben White Boulevard, Austin, TX 78741 (US).

(81) Designated States *(national)*: CN, JP, KR, SG.

(84) Designated States *(regional)*: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published:
— *With international search report.*
— *Before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments.*

(54) Title: FAST MULTITHREADING FOR CLOSELY COUPLED MULTIPROCESSORS



(57) Abstract: A processor architecture containing multiple closely coupled processors in a form of symmetric multiprocessing system is provide. The special coupling mechanism allows it to speculatively execute multiple threads in parallel very efficiently. Generally, the operating system is responsible for scheduling various threads of execution among the available processors in a multiprocessor system. One problem with parallel multithreading is that the overhead involved in scheduling the threads for execution by the operating system is such that shorter segments of code cannot efficiently take advantage of parallel multithreading. Consequently, potential performance gains from parallel multithreading are not attainable. Additional circuitry is included in a form of symmetrical multiprocessing system which enables the scheduling and speculative execution of multiple threads on multiple processors without the involvement and inherent overhead of the operating system. Advantageously, parallel multithreaded execution is more efficient and performance may be improved.

WO 01/33352 A1

TITLE: FAST MULTITHREADING FOR CLOSELY COUPLED MULTIPROCESSORS

BACKGROUND OF THE INVENTION

1. Technical Field

This invention is related to the field of microprocessors and, more particularly, to multithreading in multiprocessors.

2. Background Art

Computer systems employing multiple processing units hold a promise of economically accommodating performance capabilities that surpass those of current single-processor based systems. Within a multiprocessing environment, rather than concentrating all the processing for an application in a single processor, tasks are divided into groups or "threads" that can be handled by separate processors. The overall processing load is thereby distributed among several processors, and the distributed tasks may be executed simultaneously in parallel. The operating system software divides various portions of the program code into the separately executable threads, and typically assigns a priority level to each thread.

Superscalar microprocessors achieve high performance by executing multiple instructions per clock cycle and by choosing the shortest possible clock cycle consistent with the design. As used herein, the term "clock cycle" refers to an interval of time accorded to various stages of an instruction processing pipeline within the microprocessor. Storage devices (e.g. registers and arrays) capture their values according to the clock cycle. For example, a storage device may capture a value according to a rising or falling edge of a clock signal defining the clock cycle. The storage device then stores the value until the subsequent rising or falling edge of the clock signal, respectively. The term "instruction processing pipeline" is used herein to refer to the logic circuits employed to process instructions in a pipelined fashion. Although the pipeline may be divided into any number of stages at which portions of instruction processing are performed, instruction processing generally comprises fetching the instruction, decoding the instruction, executing the instruction, and storing the execution results in the destination identified by the instruction.

An important feature of microprocessors is the degree to which they can take advantage of parallelism. Parallelism is the execution of instructions in parallel, rather than serially. Superscalar processors are able to identify and utilize fine grained instruction level parallelism by executing certain instructions in parallel. However, this type of parallelism is limited by data dependencies between instructions. Further, as mentioned above, computer systems which contain more than one processor may improve performance by dividing the workload presented by the computer processes. By identifying higher levels of parallelism, multi-processor computer systems may execute larger segments of code, or threads, in parallel on separate processors. Because microprocessors and operating systems cannot identify these segments of code which are amenable to parallel multithreaded execution, they are identified by the application code itself. Generally, the operating system is responsible for scheduling the various threads of execution among the available processors in a multi-processor system.

One problem with parallel multithreading is that the overhead involved in scheduling the threads for execution by the operating system is such that shorter segments of code cannot efficiently take advantage of parallel multithreading. Consequently, potential performance gains from parallel multithreading are not attainable.

DISCLOSURE OF INVENTION

The problems outlined above are in large part solved by a microprocessor and method as described herein. Additional circuitry is included in a form of symmetrical multiprocessing system which enables the scheduling and speculative execution of multiple threads on multiple processors without the involvement and inherent overhead of the operating system. Advantageously, parallel multithreaded execution is more efficient and performance is improved.

Broadly speaking, a multiprocessor computer is contemplated comprising a plurality of processors, wherein said processors include a register file, a reorder buffer and circuitry to support speculative multithreaded execution. In addition, the multiprocessor computer includes one or more reorder buffer tag translation buffers and a thread control device. The thread control device is configured to store and transmit instructions between the processors. The thread control device and instructions support parallel speculative multithreaded execution.

In addition, a method is contemplated which comprises performing thread setup for execution of a second thread on a second processor, wherein the setup comprises a first processor conveying setup instructions to a second processor, where the setup instructions are speculatively executed on the second processor. A startup instruction is conveyed from the first processor to the second processor which begins speculative execution of the second thread on the second processor. The second processor begins speculative execution of the second thread in parallel with the execution of a thread on the first processor, in response to receiving the startup instruction. Execution of the second thread is terminated, in response to retiring a termination instruction in the second processor. Finally, the results of the execution of the second thread are conveyed to the first processor, in response to the second processor receiving a retrieve result instruction, where the retrieve result instruction is speculatively executed by the second processor.

BRIEF DESCRIPTION OF DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

Fig. 1 is a block diagram of one embodiment of a microprocessor.

Fig. 2 is a block diagram of one embodiment of a multiprocessor computer.

Fig. 3A is a diagram showing two microprocessors and a thread control device.

Fig. 3B is a flowchart illustrating thread setup, execution and completion.

Fig. 4 is a diagram showing two microprocessors and a thread control device.

Fig. 5 is a chart showing two threads of instructions and the use of a Sync instruction.

Fig. 6 shows a synchronous implementation of synchronization logic.

Fig. 7 shows an asynchronous implementation of synchronization logic.

Fig. 8 shows an instruction sequence representing an asynchronous implementation of synchronization logic.

Fig. 9 shows one embodiment of an interprocessor communication unit and a thread control device.

Fig. 10 is a block diagram of one embodiment of a computer system including the multiprocessor computer shown in figure 2.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

MODE(S) FOR CARRYING OUT THE INVENTION

Processor Overview

Turning now to Fig. 1, a block diagram of one embodiment of a processor 10 is shown. Other embodiments are possible and contemplated. As shown in Fig. 1, processor 10 includes a prefetch/predecode unit 12, a branch prediction unit 14, an instruction cache 16, an instruction alignment unit 18, a plurality of decode units 20A-20C, a plurality of reservation stations 22A-22C, a plurality of functional units 24A-24C, a load/store unit 26, a data cache 28, a register file 30, a reorder buffer 32, an MROM unit 34, an interprocessor communication unit 300, and a bus interface unit 37. Elements referred to herein with a particular reference number followed by a letter will be collectively referred to by the reference number alone. For example, decode units 20A-20C will be collectively referred to as decode units 20.

Prefetch/predecode unit 12 is coupled to receive instructions from bus interface unit 37, and is further coupled to instruction cache 16 and branch prediction unit 14. Similarly, branch prediction unit 14 is coupled to instruction cache 16. Still further, branch prediction unit 14 is coupled to decode units 20 and functional units 24. Instruction cache 16 is further coupled to MROM unit 34 and instruction alignment unit 18. Instruction alignment unit 18 is in turn coupled to decode units 20. Each decode unit 20A-20C is coupled to load/store unit 26 and to respective reservation stations 22A-22C. Reservation stations 22A-22C are further coupled to respective functional units 24A-24C. Additionally, decode units 20 and reservation stations 22 are coupled to register file 30 and reorder buffer 32. Functional units 24 are coupled to load/store unit 26, register file 30, and reorder buffer 32 as well. Data cache 28 is coupled to load/store unit 26 and to bus interface unit 37. Bus interface unit 37 is further coupled to an L2 interface to an L2 cache and a bus. Interprocessor communication unit 300 is coupled to reorder buffer 32 and result bus 38. Finally, MROM unit 34 is coupled to decode units 20.

Instruction cache 16 is a high speed cache memory provided to store instructions. Instructions are fetched from instruction cache 16 and dispatched to decode units 20. In one embodiment, instruction cache 16 is configured to store up to 64 kilobytes of instructions in a 2 way set associative structure having 64 byte lines (a byte comprises 8 binary bits). Alternatively, any other desired configuration and size may be employed. For example, it is noted that instruction cache 16 may be implemented as a fully associative, set associative, or direct mapped configuration.

Instructions are stored into instruction cache 16 by prefetch/predecode unit 12. Instructions may be prefetched prior to the request thereof from instruction cache 16 in accordance with a prefetch scheme. A variety of prefetch schemes may be employed by prefetch/predecode unit 12. As prefetch/predecode unit 12 transfers instructions to instruction cache 16, prefetch/predecode unit 12 generates three predecode bits for each byte of the instructions: a start bit, an end bit, and a functional bit. The predecode bits form tags indicative of the boundaries of each instruction. The predecode tags may also convey additional information such as whether a given instruction can be decoded directly by decode units 20 or whether the instruction is executed by invoking a microcode procedure controlled by MROM unit 34, as will be described in greater detail below. Still further, prefetch/predecode unit 12 may be configured to detect branch instructions and to store branch prediction information corresponding to the branch instructions into branch prediction unit 14. Other embodiments may employ any suitable predecode scheme.

One encoding of the predecode tags for an embodiment of processor 10 employing a variable byte length instruction set will next be described. A variable byte length instruction set is an instruction set in which different instructions may occupy differing numbers of bytes. An exemplary variable byte length instruction set employed by one embodiment of processor 10 is the x86 instruction set.

In the exemplary encoding, if a given byte is the first byte of an instruction, the start bit for that byte is set. If the byte is the last byte of an instruction, the end bit for that byte is set. Instructions which may be directly decoded by decode units 20 are referred to as "fast path" instructions. The remaining x86 instructions are referred to as MROM instructions, according to one embodiment. For fast path instructions, the functional bit is set for each prefix byte included in the instruction, and cleared for other bytes. Alternatively, for MROM instructions, the functional bit is cleared for each prefix byte and set for other bytes. The type of instruction may be determined by examining the functional bit corresponding to the end byte. If that functional bit is clear, the instruction is a fast path instruction. Conversely, if that functional bit is set, the instruction is an MROM instruction. The opcode of an instruction may thereby be located within an instruction which may be directly decoded by decode units 20 as the byte associated with the first clear functional bit in the instruction. For example, a fast path instruction including two prefix bytes, a Mod R/M byte, and an immediate byte would have start, end, and functional bits as follows:

Start bits	10000
End bits	00001
Functional bits	11000

MROM instructions are instructions which are determined to be too complex for decode by decode units 20. MROM instructions are executed by invoking MROM unit 34. More specifically, when an MROM instruction is encountered, MROM unit 34 parses and issues the instruction into a subset of defined fast path instructions to effectuate the desired operation. MROM unit 34 dispatches the subset of fast path instructions to decode units 20.

Processor 10 employs branch prediction in order to speculatively fetch instructions subsequent to conditional branch instructions. Branch prediction unit 14 is included to perform branch prediction operations. In one embodiment, branch prediction unit 14 employs a branch target buffer which caches up to two branch target addresses and corresponding taken/not taken predictions per 16 byte portion of a cache line in instruction cache 16. The branch target buffer may, for example, comprise 2048 entries or any other suitable number of entries. Prefetch/predecode unit 12 determines initial branch targets when a particular line is predecoded.

Subsequent updates to the branch targets corresponding to a cache line may occur due to the execution of instructions within the cache line. Instruction cache 16 provides an indication of the instruction address being fetched, so that branch prediction unit 14 may determine which branch target addresses to select for forming a branch prediction. Decode units 20 and functional units 24 provide update information to branch prediction unit 14. Decode units 20 detect branch instructions which were not predicted by branch prediction unit 14. Functional units 24 execute the branch instructions and determine if the predicted branch direction is incorrect. The branch direction may be "taken", in which subsequent instructions are fetched from the target address of the branch instruction. Conversely, the branch direction may be "not taken", in which subsequent instructions are fetched from memory locations consecutive to the branch instruction. When a mispredicted branch instruction is detected, instructions subsequent to the mispredicted branch are discarded from the various units of processor 10. In an alternative configuration, branch prediction unit 14 may be coupled to reorder buffer 32 instead of decode units 20 and functional units 24, and may receive branch misprediction information from reorder buffer 32. A variety of suitable branch prediction algorithms may be employed by branch prediction unit 14.

Instructions fetched from instruction cache 16 are conveyed to instruction alignment unit 18. As instructions are fetched from instruction cache 16, the corresponding predecode data is scanned to provide information to instruction alignment unit 18 (and to MROM unit 34) regarding the instructions being fetched. Instruction alignment unit 18 utilizes the scanning data to align an instruction to each of decode units 20. In one embodiment, instruction alignment unit 18 aligns instructions from three sets of eight instruction bytes to decode units 20. Decode unit 20A receives an instruction which is prior to instructions concurrently received by decode units 20B and 20C (in program order). Similarly, decode unit 20B receives an instruction which is prior to the instruction concurrently received by decode unit 20C in program order.

Decode units 20 are configured to decode instructions received from instruction alignment unit 18. Register operand information is detected and routed to register file 30 and reorder buffer 32. Additionally, if the instructions require one or more memory operations to be performed, decode units 20 dispatch the memory operations to load/store unit 26. Each instruction is decoded into a set of control values for functional units 24, and these control values are dispatched to reservation stations 22 along with operand address information and displacement or immediate data which may be included with the instruction. In one particular embodiment, each instruction is decoded into up to two operations which may be separately executed by functional units 24A-24C.

Processor 10 supports out of order execution, and thus employs reorder buffer 32 to keep track of the original program sequence for register read and write operations, to implement register renaming, to allow for speculative instruction execution and branch misprediction recovery, and to facilitate precise exceptions. A temporary storage location within reorder buffer 32 is reserved upon decode of an instruction that involves the update of a register to thereby store speculative register states. If a branch prediction is incorrect, the results of speculatively-executed instructions along the mispredicted path can be invalidated in the buffer before they are written to register file 30. Similarly, if a particular instruction causes an exception, instructions subsequent to the particular instruction may be discarded. In this manner, exceptions are "precise" (i.e. instructions subsequent to the particular instruction causing the exception are not completed prior to the exception). It is noted that a particular instruction is speculatively executed if it is executed prior to instructions which precede the particular instruction in program order. Preceding instructions may be a branch instruction or an exception-causing instruction, in which case the speculative results may be discarded by reorder buffer 32.

The instruction control values and immediate or displacement data provided at the outputs of decode units 20 are routed directly to respective reservation stations 22. In one embodiment, each reservation station 22 is capable of holding instruction information (i.e., instruction control values as well as operand values, operand tags and/or immediate data) for up to five pending instructions awaiting issue to the corresponding functional unit. It is noted that for the embodiment of Fig. 1, each reservation station 22 is associated with a dedicated functional unit 24. Accordingly, three dedicated "issue positions" are formed by reservation stations 22 and functional units 24. In other words, issue position 0 is formed by reservation station 22A and functional unit 24A. Instructions aligned and dispatched to reservation station 22A are executed by functional unit 24A. Similarly, issue position 1 is formed by reservation station 22B and functional unit 24B; and issue position 2 is formed by reservation station 22C and functional unit 24C.

Upon decode of a particular instruction, if a required operand is a register location, register address information is routed to reorder buffer 32 and register file 30 simultaneously. In one embodiment, reorder buffer 32 includes a future file which receives operand requests from decode units as well. Those of skill in the art will appreciate that the x86 register file includes eight 32 bit real registers (i.e., typically referred to as EAX, EBX, ECX, EDX, EBP, ESI, EDI and ESP). In embodiments of processor 10 which employ the x86 processor architecture, register file 30 comprises storage locations for each of the 32 bit real registers. Additional storage locations may be included within register file 30 for use by MROM unit 34. Reorder buffer 32 contains temporary storage locations for results which change the contents of these registers to thereby allow out of order execution. A temporary storage location of reorder buffer 32 is reserved for each instruction which, upon decode, is determined to modify the contents of one of the real registers. Therefore, at various points during execution of a particular program, reorder buffer 32 may have one or more locations which contain the speculatively executed contents of a given register. If following decode of a given instruction it is determined that reorder buffer 32 has a previous location or locations assigned to a register used as an operand in the given instruction, the reorder buffer 32 forwards to the corresponding reservation station either: 1) the value in the most recently assigned location, or 2) a tag for the most recently assigned location if the value has not yet been produced by the functional unit that will eventually execute the previous instruction. If reorder buffer 32 has a location reserved for a given register, the operand value (or reorder buffer tag) is provided from reorder buffer 32 rather than from register file 30. If there is no location reserved for a required register in reorder buffer 32, the value is taken directly from register file 30. If the operand corresponds to a memory location, the operand value is provided to the reservation station through load/store unit 26.

In one particular embodiment, reorder buffer 32 is configured to store and manipulate concurrently decoded instructions as a unit. This configuration will be referred to herein as "line-oriented". By manipulating several instructions together, the hardware employed within reorder buffer 32 may be simplified. For example, a line-oriented reorder buffer included in the present embodiment allocates storage sufficient for instruction information pertaining to three instructions (one from each decode unit 20) whenever one or more instructions are issued by decode units 20. By contrast, a variable amount of storage is allocated in conventional reorder buffers, dependent upon the number of instructions actually dispatched. A comparatively larger number of logic gates may be required to allocate the variable amount of storage. When each of the concurrently decoded instructions has executed, the instruction results are stored into register file 30 simultaneously. The storage is then free for allocation to another set of concurrently decoded instructions. Additionally, the amount of control logic circuitry employed per instruction is reduced because the control logic is amortized over several concurrently decoded instructions. A reorder buffer tag identifying a particular instruction may be divided into two fields: a line tag and an offset tag. The line tag identifies the set of concurrently decoded instructions including the particular instruction, and the offset tag identifies which instruction within the set corresponds to the particular instruction. It is noted that storing instruction results into register file 30 and freeing the corresponding storage is referred to as "retiring" the instructions. It is further noted that any reorder buffer configuration may be employed in various embodiments of processor 10, including using a future file to store the speculative state of register file 30.

As noted earlier, reservation stations 22 store instructions until the instructions are executed by the corresponding functional unit 24. An instruction is selected for execution if: (i) the operands of the instruction have been provided; and (ii) the operands have not yet been provided for instructions which are within the same reservation station 22A-22C and which are prior to the instruction in program order. It is noted that when an instruction is executed by one of the functional units 24, the result of that instruction is passed directly to any reservation stations 22 that are waiting for that result at the same time the result is passed to update reorder buffer 32 (this technique is commonly referred to as "result forwarding"). An instruction may be selected for execution and passed to a functional unit 24A-24C during the clock cycle that the associated result is forwarded. Reservation stations 22 route the forwarded result to the functional unit 24 in this case. In embodiments in which instructions may be decoded into multiple operations to be executed by functional units 24, the operations may be scheduled separately from each other.

In one embodiment, each of the functional units 24 is configured to perform integer arithmetic operations of addition and subtraction, as well as shifts, rotates, logical operations, and branch operations. The operations are performed in response to the control values decoded for a particular instruction by decode units 20. It is noted that a floating point unit (not shown) may also be employed to accommodate floating point operations. The floating point unit may be operated as a coprocessor, receiving instructions from MROM unit 34 or reorder buffer 32 and subsequently communicating with reorder buffer 32 to complete the instructions. Additionally, functional units 24 may be configured to perform address generation for load and store memory operations performed by load/store unit 26. In one particular embodiment, each functional unit 24 may comprise an address generation unit for generating addresses and an execute unit for performing the remaining functions. The two units may operate independently upon different instructions or operations during a clock cycle.

Each of the functional units 24 also provides information regarding the execution of conditional branch instructions to the branch prediction unit 14. If a branch prediction was incorrect, branch prediction unit 14 flushes instructions subsequent to the mispredicted branch that have entered the instruction processing pipeline, and causes fetch of the required instructions from instruction cache 16 or main memory. It is noted that in such situations, results of instructions in the original program sequence which occur after the mispredicted branch instruction are discarded, including those which were speculatively executed and temporarily stored in load/store unit 26 and reorder buffer 32. It is further noted that branch execution results may be provided by functional units 24 to reorder buffer 32, which may indicate branch mispredictions to functional units 24.

Results produced by functional units 24 are sent to reorder buffer 32 if a register value is being updated, and to load/store unit 26 if the contents of a memory location are changed. If the result is to be stored in a register, reorder buffer 32 stores the result in the location reserved for the value of the register when the instruction was decoded. A plurality of result buses 38 are included for forwarding of results from functional units 24 and load/store unit 26. Result buses 38 convey the result generated, as well as the reorder buffer tag identifying the instruction being executed.

Load/store unit 26 provides an interface between functional units 24 and data cache 28. In one embodiment, load/store unit 26 is configured with a first load/store buffer having storage locations for data and address information for pending loads or stores which have not accessed data cache 28 and a second load/store buffer having storage locations for data and address information for loads and stores which have access data cache 28. For example, the first buffer may comprise 12 locations and the second buffer may comprise 32 locations. Decode units 20 arbitrate for access to the load/store unit 26. When the first buffer is full, a decode unit must wait until load/store unit 26 has room for the pending load or store request information. Load/store unit 26 also performs dependency checking for load memory operations against pending store memory operations to ensure that data coherency is maintained. A memory operation is a transfer of data between processor 10 and the main memory subsystem. Memory operations may be the result of an instruction which utilizes an operand stored in memory, or may be the result of a load/store instruction which causes the data transfer but no other operation. Additionally, load/store unit 26 may include a special register storage for special registers such as the segment registers and other registers related to the address translation mechanism defined by the x86 processor architecture.

Data cache 28 is a high speed cache memory provided to temporarily store data being transferred between load/store unit 26 and the main memory subsystem. In one embodiment, data cache 28 has a capacity of storing up to 64 kilobytes of data in an two way set associative structure. It is understood that data cache 28 may be implemented in a variety of specific memory configurations, including a set associative configuration, a fully associative configuration, a direct-mapped configuration, and any suitable size of any other configuration.

Interprocessor communication unit 300 provides an interface for communicating with another processor in a multithreaded multiprocessor configuration. In one embodiment, interprocessor communication unit 300 may include a reservation station for temporarily storing instructions to be executed. In addition, in one embodiment, interprocessor communication unit 300 interfaces to a thread control device which facilitates multithread related communications between processors.

In one particular embodiment of processor 10 employing the x86 processor architecture, instruction cache 16 and data cache 28 are linearly addressed and physically tagged. The linear address is formed from the offset specified by the instruction and the base address specified by the segment portion of the x86 address translation mechanism. Linear addresses may optionally be translated to physical addresses for accessing a main memory. The linear to physical translation is specified by the paging portion of the x86 address translation mechanism. The physical address is compared to the physical tags to determine a hit/miss status.

Bus interface unit 37 is configured to communicate between processor 10 and other components in a computer system via a bus. For example, the bus may be compatible with the EV-6 bus developed by Digital Equipment Corporation. Alternatively, any suitable interconnect structure may be used including packet-based, unidirectional or bi-directional links, etc. An optional L2 cache interface may be employed as well for interfacing to a level two cache.

Symmetric Multiprocessing

Fig. 2 is a block diagram of one embodiment of a multiprocessor computer 100 including a plurality of processing units 12A-12B, a thread control device 300, a bus bridge 30 and a memory 20. Each processing unit 12A-12B includes a processing core 14A-14B, an L1 cache memory 16A-16B, and a bus interface 18A-18B, respectively. The processing units 12A-12B are coupled to a main memory 20 via a system bus 22.

The multiprocessor computer 100 of fig. 2 is symmetrical in the sense that all processing units 12A-12B share the same memory space (i.e., main memory 20) and access the memory space using the same address mapping. The multiprocessing system 100 is further symmetrical in the sense that all processing units 12A-12B share equal access to the same I/O subsystem.

In general, a single copy of the operating system software as well as a single copy of each user application file is stored within main memory 20. Each processing unit 12A-12B executes from these single copies of the operating system and user application files. Although processing cores 14A-14B may be executing code simultaneously, it is noted that only one of the processing units 12A-12B may assume mastership of the system bus 22 at a given time. Thus, a bus arbitration mechanism, bus bridge 30, is provided to arbitrate concurrent bus requests of two or more processing units and to grant mastership to one of the processing units based on a predetermined arbitration algorithm. A variety of bus arbitration techniques are well-known.

The high speed cache memory 16A-16B of each processing unit 12A-12B, respectively, stores data most recently accessed by the respective processing unit along with address tags that indicate the main memory address to which the associated data corresponds. Since programs tend to execute the same sections of code and access the same data structures repeatedly, many of the locations accessed will already be stored in the cache if the cache is sufficiently large.

The cache mechanisms provide two significant benefits. First, because the caches are implemented with high-speed memory and can be accessed without bus arbitration and buffer delays, an access to a location stored in a respective cache is much faster than a main memory access. Second, because an access to a location stored in the respective cache does not require access to the system bus, the bus utilization of each processor is greatly reduced. The system bus is therefore available to service other requested transactions. Typically, the higher the "hit rate", the better the overall system performance. The hit rate is the percentage of accesses by a particular processing core that are to locations already stored in the cache. Well designed systems with moderately large caches can achieve hit rates of over ninety percent.

An important consideration with respect to multiprocessing systems that employ cache memories is data coherency. Since multiple copies of the data (and instructions) stored by main memory 20 may concurrently reside in one or more of the cache memories 16A-16B, a specialized mechanism must be employed to maintain the integrity of data in the event that one of the memory subsystems is updated (i.e., written with new data). For example, consider a situation wherein a particular section of data is updated within cache memory 16A by processing core 14A but is not updated within the corresponding section of main memory 20. If processing core 14B subsequently accesses the same section of code, there must be some reliable mechanism to track which section is up-to-date and which section is no longer valid to ensure that processing core 14B accesses the proper data. A variety of techniques have therefore been developed with the goal of efficiently maintaining cache coherency, including those based on so-called write-through and write-back techniques. Various cache coherency techniques are described within a host of publications of the known prior art, and are not discussed further herein.

General Operation

Figs. 3A and 3B illustrate the general operation of multithreaded multiprocessing. In fig. 3A, interconnection between two processors 12A-12B and a thread control device 300 is shown. Fig. 3B is a flowchart illustrating the general operation of the multiprocessor computer shown in fig. 3A. Each processor 12A-12B includes a processing core 14A-14B, an L1 cache 16A-16B, and an interprocessor communication unit (hereinafter ICU) 320A-320B. Also shown is thread control device 300 which includes first-in-first-out (hereinafter FIFO) buffers 310A-310B, and synchronization logic 314.

The ICUs 320 of each processor 12 are coupled to thread control device 300 which facilitates communication between processors 12A and 12B. In general, one processor 12A serves as master and the other processor 12B serves as slave. The master processor 12A runs all single threaded code, sets up and starts thread execution on the slave processor 12B and consolidates execution results following thread execution. In general, processing core 14A executes single threaded code (block 330) until a multithread setup instruction is encountered. When processing core 12A encounters a multithread setup instruction (block 332), processing core 12A conveys thread setup instructions to ICU 320A which conveys them to FIFO 1 310A. ICU 320B retrieves instructions from FIFO 1 310A and transfers them to processing core 14B. Subsequently, master processor 12A conveys a thread 2 startup instruction (block 334) to ICU 320A which places the instruction into FIFO 1 310A. ICU 320B retrieves the thread startup instruction from FIFO 1 310A and transfers it to processing core 14B. Processing core 14B then begins fetching and executing the thread 2 code (block 338). Upon execution and retirement of a JOIN instruction (blocks 340 and 342) by both processors 12, slave processor 12B terminates execution of thread 2 and single threaded execution resumes with master processor 12A. Master processor 12A may then convey another instruction to processor 12B which causes slave processor 12B to convey thread 2 execution results to master processor 12A via FIFO 310B. Master processor 12A may then consolidate execution results from the separate threads (block 344) and continue normal execution (block 346). To summarize, master processor 12A sets up a second thread for execution on slave processor 12B. Both the master 12A and slave 12B processors execute threads in parallel. Master processor 12A then obtains the second thread execution results from the slave processor.

Detailed Description of Operation

Fig. 4 shows one embodiment of a multithreaded multiprocessor. Included in Fig. 4 are portions of processors 12A and 12B, thread control device 300, and bus 22. Processors 12 include an ICU 320, register file 30, reorder buffer 32, system bus 38 and bus interface unit 37. In addition, ICU 320 includes a reservation station 402 and a reorder buffer tag translation buffer (hereinafter RTB) 400. Thread control device 300 includes two FIFOs 310 and synchronization logic 314. ICUs 320 are coupled to reorder buffer 32 and system bus 38. Register file 30 and bus interface unit 37 are coupled to bus 22. Bus interface unit 37 is also coupled to bus 22. Thread control device 300 is coupled to ICU 320.

In one embodiment of the multithreaded multiprocessor shown in Fig. 4, the five following instructions are used for the control of threads: Wr2Proc, RdFrProc, Fork, Join, Sync. Table 1 below gives a brief description of the purpose of each instruction.

Instruction	Syntax	Purpose
Wr2Proc	Wr2Proc PNo, destreg, srcreg	To move data from the register file of the current processor to the register file of another processor. PNo = number of destination processor. destreg = register identifier on destination processor. srcreg = register identifier on current processor (source).
RdFrProc	Rd2Proc PNo, destreg, srcreg	To move data from the register file of another processor to the register file of the current processor. PNo = number of destination processor. destreg = register identifier on current processor (destination). srcreg = register identifier on source processor.
Fork	PNo, ThrdAddr	Starts speculative execution of a thread on another processor. PNo = number of processor to start thread on. ThrdAddr = address of thread code.
Join	Join	Ends execution of thread code.
Sync	Sync	Serializes execution until all processors have reached the synchronization point.

Table 1 – Thread Control Instructions

Briefly, master processor 12A sets up a second thread for execution by conveying Wr2Proc instructions to slave processor 12B. Wr2Proc instructions move data from the register file of the master processor to the slave processor. To start execution of the second thread, the Fork instruction is used. When the master processor 12A conveys a Fork instruction to slave processor 12B, slave processor 12B places the Fork instruction in its reorder buffer 32B and begins fetching the instructions for the second thread from the thread address conveyed with the Fork instruction. Execution of the second thread terminates upon execution of a Join instruction. The ability to speculatively execute thread instructions is important as is discussed below. Support for speculative execution is discussed next, followed by a more detailed discussion of the overall mechanism of thread setup and execution.

Support for Speculative Thread Execution

State of the art superscalar processors have large instruction windows. Consequently, to wait for a Fork instruction to retire before thread startup may result in significant delays. To allow optimal thread startup, the mechanism should allow for speculative startup of threads. This allows the second thread to startup and execute in the slave processor long before the Fork instruction retires in the master processor. Advantageously, performance of the multithreaded multiprocessor is improved.

To support speculative thread execution, a reorder buffer tag translation buffer (RTB) 400 is included in each processor 12 which maps the location of an instruction in one processor reorder buffer to the location of the same instruction in another processor reorder buffer. In one embodiment, the RTB 400 has the same number of entries as the reorder buffer 32. The RTB is addressed with the reorder buffer 32 tag of a first processor and the addressed entry of the RTB 400 contains the corresponding reorder buffer tag 32 of the second processor.

To enable speculative startup and execution of threads, the Wr2Proc and Fork instructions are processed speculatively. When Wr2Proc and Fork are dispatched to the master processor ICU 320A, they are also sent to the slave processor 12B via FIFO 1 310A. Two possibilities may exist when the Wr2Proc instruction is dispatched to the master ICU 320A: data is available or data is not available. If data is available, the instruction, reorder buffer 32A tag of the instruction, and an "add instruction" are sent to the slave processor. The instruction and data are inserted into the reorder buffer 32B of the slave processor and the entry is marked as having valid data. In addition, the RTB 400B of the slave is updated to indicate the correspondence between the reorder buffer 32A entry and the reorder buffer 32B entry. If data is not available upon dispatch to the master processor ICU 320A, the instructions remain in the ICU reservation station 402A of the master processor ICU 320A until data is available. However, the instruction is still sent to the slave processor 12B. The instructions are inserted into the reorder buffer 32B of the slave processor, the entries in the slave processor reorder buffer 32B are marked as not having valid data, and the RTB 400B is updated as described above. Instructions in slave processor reorder buffer 32B that are marked as not having valid data may not execute until data is available. When data becomes available for an instruction waiting in the reservation station 402A of the master processor 12A, the instruction is removed from the reservation station 402A of the master processor 12A and issues to ICU 320A. When the ICU 320A receives the instruction, the data and the reorder buffer 32A tag of the master processor, along with a "data update" command are sent to the slave processor 12B. The slave processor 12B translates the reorder buffer 32A tag using the RTB 400B to identify the tag of the instruction in the slave processor 12B reorder buffer 32B. The corresponding reorder buffer 32B tag is then used to insert the data into the correct reorder buffer 32B entry. The instruction in the slave processor reorder buffer 32B is now marked as having valid data and dependent instructions in slave processor 12B may issue.

Thread Setup and Thread Startup

To setup a second thread for execution on another processor, master processor 12A conveys speculative Wr2Proc instructions to slave processor 12B via FIFO 1 310A. In addition to the Wr2Proc instruction, master processor 12A conveys the reorder buffer 32A tag of the instruction to slave processor 12B. The instructions are placed into the reorder buffer 32B of the slave processor. If the data for the Wr2Proc instruction is available, it is placed in reorder buffer 32B as well. Otherwise, the reorder buffer 32B entry is marked as not having valid data. In parallel, the RTB 400B of the slave processor is updated by placing the tag of the slave processor reorder buffer 32B in the location indexed by the tag of the master processor reorder buffer 32A. If the reorder buffer 32B of the slave processor 12B is full, no instructions will be received from the master processor 12A. When space is available in slave processor reorder buffer 12B, master processor 12A receives an indication and pending instructions may be sent.

After setup of the second thread is complete, the Fork instruction is used to start execution of the second thread. When a Fork instruction is encountered by the master processor, it is sent to the slave processor 12B via thread control device 300 as described above. If the slave processor 12B is already running a thread, the Fork instruction is ignored. An indication is made by the slave processor 12B as to the success of the Fork operation. Such an indication may be made by various means, such as setting an ordinary processor flag which may be checked by the master processor 12A. Other embodiments may use other means to ensure an attempt to start a second thread is not made while another thread is already is running, eliminating the need for setting and checking of flags. If the slave processor 12B is not already running a thread, the Fork instruction is placed in reorder buffer 32B and the slave processor begins fetching instructions for the second thread. The newly fetched thread instructions are placed in reorder buffer 32B behind the Fork instruction and are dispatched to the processor functional units for execution. Those instructions which have no outstanding dependencies on the Wr2Proc instructions may issue. Advantageously, many nondependent instructions may issue in the slave processor 12B before the Fork instruction has retired in the master processor 12A. The mechanism of executing the thread instructions after the Fork instruction in the slave processor 12B is identical to a regular superscalar processor.

Branch Misprediction and Correction

When a branch misprediction is detected in the master processor 12A, all entries in the reorder buffer 32A following the branch are invalidated. If Wr2Proc or Fork instructions were mispredicted, these need to be invalidated in the slave processor 12B as well. When master processor 12A invalidates entries in reorder buffer 32A following a mispredicted branch, it detects the first Wr2Proc or Fork instruction following the mispredicted branch and sends the reorder buffer 32A tag of that instruction along with an invalidation request to the ICU 320A. This tag and request are then conveyed to the slave processor 12B where the master processor reorder buffer 32A tag is translated by the RTB 400B to obtain the reorder buffer 32B tag of the slave processor 12B. The resulting reorder buffer 32B tag is then used to invalidate that entry and all following entries in the reorder buffer 32B. If a Fork instruction is encountered during invalidation of instructions in reorder buffer 32B, speculative execution in slave processor 12B stops.

Ending Second Thread Execution and Retirement

The end of the second thread executing on the slave processor 12B is indicated by a Join instruction. A Join instruction is also used in the thread running on the master processor 12A. When both the master and slave processors 12 have retired the Join instruction, the slave processor 12B stops executing and execution continues only in the master processor 12A. When a slave processor 12B retires a Join instruction, it signals this retirement to the master processor 12A and its reorder buffer 32B is cleared. The slave processor 12B then stops execution and waits for the next Fork instruction. When the master processor 12A has received an indication that the slave processor 12B has retired the Join instruction, the master processor 12A marks its Join instruction as completed and ready to retire.

Once both processors 12 have retired the Join instruction, the master processor 12A may access the register file 30B of the slave processor 12B to obtain the execution results of the second thread. Access to the slave register file 30B is obtained by use of the RdFrProc instruction. The RdFrProc instruction is dispatched by the master processor 12A to the ICU 320A where it waits in the reservation station 402A until it is at the front and then it is issued. The RdFrProc command is then sent to the slave processor 12B. Execution of the RdFrProc command in the slave processor 12B reads the contents of the specified register and conveys the results back to the master processor 12A via FIFO 2 310B. The RdFrProc command in the master processor, which is still in the execute phase, retrieves the result and places it on the result bus 38A. Normal instruction execution then continues.

The RdFrProc instruction may issue before the Join instruction retires, but may not retire until after the Join instruction retires. Because synchronization is not performed prior to a RdFrProc instruction, a Join instruction must precede that instruction in both the master and slave threads. If a mispredicted branch occurs and RdFrProc instructions are mispredicted, the instruction may still receive the data but the result is discarded by the master processor reorder buffer 32A. Advantageously, the above mechanism enables speculative execution of the RdFrProc instruction.

Retiring Wr2Proc and Fork Instructions

When a Wr2Proc or Fork instruction retires in the master processor reorder buffer 32A, the reorder buffer 32A tag is sent to the slave processor where it is translated by RTB 400B to the reorder buffer 32B tag of the slave processor. Those instructions which have retired in the master processor 12A may then be retired from the slave processor 12B as well.

The retirement of Wr2Proc and Fork instructions may be handled in a variety of ways. One method involves a retirement command and the second involves a Fork commit command. In order to implement branch misprediction recovery as described above, the Wr2Proc and Fork command cannot retire in the slave processor reorder buffer 32B until they have retired in the master processor reorder buffer 32A. Consequently, reorder buffer 32A must notify reorder buffer 32B when such a retirement occurs. One way of making this notification is to send a retire command along with the reorder buffer 32A tag to the slave processor 12B whenever a Wr2Proc or Fork instruction retires in the master reorder buffer 32A. The master reorder buffer 32A tag is then translated by RTB 400B in the slave processor 12B to obtain the slave reorder buffer 32B tag. The resulting slave reorder buffer 32B tag is then used to retire the corresponding instruction. While this method creates additional retirement traffic to slave reorder buffer 32B, the operation of this method is advantageously transparent to software, unlike the following method.

A second method of handling the retirement of Wr2Proc and Fork instructions involves the use of a Fork commit command. When a Fork instruction is retired in the master reorder buffer 32A, the reorder buffer 32A indicates to the ICU 320A that a Fork instruction has been retired along with the corresponding reorder buffer 32A tag. The ICU 320A then sends a Fork commit command and the reorder buffer 32A tag to the slave processor 12B. The reorder buffer 32A tag is then translated by RTB 400B to obtain the corresponding reorder buffer 32B tag. The resulting reorder buffer 32B tag is then used to mark the corresponding entry as completed and ready to retire. Also, all previous Wr2Proc instructions in slave reorder buffer 32B are marked completed and can now retire. When using this second method, there is a special case to consider. If for some reason Wr2Proc instructions are not followed by a Fork instruction, they may not be marked as complete and may remain in the reorder buffer 32B. Subsequent thread setups by the master processor 12A may write more Wr2Proc instructions. With each setup, less space is available in the slave reorder buffer 32B. This process may continue and ultimately result in an overflow of the slave reorder buffer 32B. To avoid this condition, the software is required to keep track of the available size of the slave reorder buffer 32B. Even though this second method may produce less traffic than the first method described above, requiring the software to continuously track the available space in the slave reorder buffer 32B is undesirable. Consequently, the first method described above which utilizes a retirement command and is transparent to software is preferred.

In one embodiment, the ICU 320 may accept multiple instructions per cycle. Reservation station 402 may accommodate multiple instructions, with the position of the instructions within reservation station 402 indicating the order in which they are to be sent to the other processor. In addition, multiple instructions may be conveyed at once to the other processor. Overall, it must be ensured that the instructions are inserted in-order in the reorder buffer 32B of the other processor.

Synchronization of Threads

Synchronization using Sync Instruction

There are times during execution of parallel threads when it may be desirable to get the threads into a determinable state. For example, if data must be exchanged between threads there needs to be a mechanism for synchronizing the execution of the threads. To enable synchronization, in one embodiment a Sync instruction and synchronization logic may be used. The synchronization mechanism used depends on the characteristics of the system. If all processors in the system have the same latencies and phases, then a highly synchronous implementation may be used. Otherwise a more flexible implementation may be used. In either case, the mechanism is transparent to the software. The same synchronization code may be used for either implementation.

Fig. 5 illustrates how a Sync instruction may be used to synchronize threads. Fig. 5 includes two columns, each representing a thread of instructions. The first column represents a thread of code running on a master processor and the second column represents a thread of code running on a slave processor. Each column in Fig. 5 is divided into rows with each cell representing a single instruction. Instruction 500 represents a label which may serve as a branch entry point from elsewhere in the code. The operation of the threads is as follows. Thread 1 code executes in parallel with thread 2 code. Thread 1 executes until it reaches the first Sync 502 instruction. Thread 2 code executes the mov instruction 510 which moves the Handle data to reg1. Thread 2 then continues execution to Sync instruction 512. When both threads have reached the first Sync instruction, 502 and 512 respectively, the Sync instructions may be retired. When both Sync instructions have exited the synchronization point (retired) execution continues. Thread 1 executes instruction 504 which reads the data from reg1 of processor 2 into reg0 of its register file. Both threads reach a second synchronization point at Sync instructions 506 and 514. When Sync instructions 506 and 514 are both ready to retire, both are retired and execution continues. Thread 1 then executes a compare instruction 507 and if they are not identical, the jump instruction 508 causes execution to continue with instruction 500. Otherwise, execution of thread 1 continues with other code. As the above example shows, a Sync instruction may not be retired until all processors are ready to retire their corresponding Sync instructions. This requires that each processor indicate when it is ready to retire a Sync instruction and each processor must know when all other processors have reached the synchronization point.

Fig. 6 shows a block diagram of a synchronous mechanism to implement the synchronization logic. Shown are synchronization logic 314, SSO signals 600, and SSI signal 602. SSO signals 600 are output from

CPUs 1-x and input to synchronization logic 314. Synchronization logic 314 outputs signal SSI 602. In the synchronous implementation shown in Fig. 6, each processor is connected to synchronization logic 314 with 2 signals: a Sync State Out (SSO) signal 600 and a Sync Signal In (SSI) 602. Each processor outputs an SSO signal 600 to indicate the state of the Sync instructions to the synchronization logic 314. In addition, each processor receives SSI signal 602 as input to detect the status of the Sync instructions in other processors. In one embodiment, the state of the SSO signal 600 is described by the following table.

SSO state	State Description
0	No Sync instruction.
1	Sync instruction ready to retire.

Table 2

Sync instructions are handled specially by the reorder buffers. In the synchronous implementation, if a Sync instruction is next to retire, then the SSO signal 600 of that processor is set to state 1 to indicate to all other processors that it has entered the synchronization point. When all SSO signals 600 input to synchronization logic 314 are set to state 1, synchronization logic 314 sets the SSI signal 602 to state 1. Only when the SSI signal 602 input to a processor is set to state 1 may that processor retire its Sync instruction. In this manner, all processors retire their Sync instructions at the same time. When a processor retires a Sync instruction, its SSO signal 600 is set to state 0 on the following clock cycle to indicate it has exited the synchronization point. The second clock cycle following retirement, the SSO signal 600 may be set to state 1 again if a Sync instruction is ready to retire. In the synchronous implementation, synchronization logic 314 may consist of an AND gate, with all SSO signals 600 as inputs and the SSI signal 602 as output. The synchronous implementation provides the fastest synchronization mechanism. However, it can only be used if the processors operate in a cycle by cycle synchronous manner.

If synchronous operation between the processors is not guaranteed, an asynchronous implementation of synchronization logic 314 may be used. Fig. 7 is a block diagram illustrating an asynchronous implementation. Fig. 7 shows synchronization logic 314, SSO signals 700 input to synchronization logic 314, and SSI signal 702 output from synchronization logic 314. In the asynchronous implementation, each processor includes a two bit SSO signal 700 to indicate the state of its Sync instruction. As in the synchronous implementation, synchronization logic 314 outputs an SSI signal 702 to each processor. In one embodiment, the state of the SSO signal 700 may be described by the following table.

SSO state	State Description
0	No Sync instruction.
1	Sync instruction ready to retire.
2	Sync instruction retired

Table 3

As in the synchronous implementation, Sync instructions are handled specially by the reorder buffers 32. In the asynchronous implementation, if a Sync instruction is next to retire, then the SSO signal of that processor is set to state 1 to indicate to the other processors that it is ready to retire its Sync instruction. When all SSO signals 700 input to synchronization logic 314 are set to state 1, synchronization logic 314 sets the SSI signal 702 to state 1. Only when the SSI signal 702 input to a processor is set to state 1 may that processor retire its Sync instruction. In this manner, all processors retire their Sync instructions at the same time. When a Sync instruction is retired, the SSO signal 700 of that processor is set to state 2 to indicate to the other processors that it is exiting the synchronization point. When a processor has exited the synchronization point and set its SSO signal 700 to state 2, it waits until all other processors have exited the synchronization point as well. When all processors have set their SSO signals 700 to state 2, SSI signal 702 is set to state 0. All processors may then detect that SSI signal 702 is set to state 0 and continue execution. After SSI signal 702 has returned to state 0, a processor may then indicate another Sync instruction is ready to retire by setting its SSI signal 700 to state 1.

The asynchronous implementation of the synchronization logic 314 may be central logic or may be included as part of one of the processors. The synchronization logic 314 implements a small state machine which may work as illustrated by fig. 8. Fig. 8 represents a continuous loop wherein the state of FSMState is continuously checked. FSMState represents the state of the Sync signals SSO in the processors. Initially, synchronization logic 314 may be in state WAIT_ALL_S1. While in state WAIT_ALL_S1, signal SSI is set to state 0. If all SSO signals are in state S1, indicating all processors are ready to retire a Sync instruction, then the next state for FSMState is ALL_S1. Otherwise, the next state for FSMState remains WAIT_ALL_S1. Once all SSO signals are in state S1, FSMState transitions to state ALL_S1 and subsequently to state WAIT_ALL_S2. In state WAIT_ALL_S2, the signal SSI is set to state 1 which allows each processor to retire its Sync instruction. While in state WAIT_ALL_S2, synchronization logic waits for all SSO signals to enter state 2. Once all processors have exited the synchronization point, as indicated by setting their SSO signals to state 2, FSMState transitions to state ALL_S2 and subsequently to state WAIT_ALL_S1 where it sets signal SSI to state 0. This implementation allows the synchronization of processors even if they do not work in lock step. While this implementation is more flexible, it has a longer latency than the synchronous implementation described above.

Synchronization using Scratchpad

Another way of exchanging data while multiple processors are processing different threads is to use a scratchpad register space. Fig. 9 shows a block diagram of an ICU 920A and TCD 900 using scratchpad registers 902 for synchronization. In addition to scratchpad registers 902, additional scratchpad instructions (hereafter SCINS) are added which access the scratchpad registers. To implement the scratchpad mechanism, scratchpad logic 902 has been added to the TCD. Shown in fig. 9 are ICU 902A and TCD 900. ICU 902A includes SCINS Read reservation station 930A, SCINS Write reservation station 932A, and FIFO reservation station 402A. TCD 900 includes FIFOs 310, synchronization logic 314, and scratchpad registers 902. ICU 902A is coupled to TCD 900.

The SCINS are locked read-modify-write (hereafter RmodW) instructions to a scratchpad register location. These instructions enable the modification of shared data during parallel thread execution. The scratchpad register may be locked on an address by address basis. The SCINS are handled by a special functional unit which may be separate from, or included in, ICU 920. The SCINS functional unit includes separate reservation stations, 930 and 932, for read and write accesses. All instructions are issued in order from the scratchpad reservation stations. Any instruction which modifies a scratchpad location with a RmodW operation is a locked instruction. SCINS are decoded into multiple operations. At least a locked Read, ALU operation, and a locked Write are generated. The mechanism works as follows:

1. A locked RmodW SCINS is decoded. From this decoded instruction, a locked Read and locked Write are dispatched to the SCINS functional unit.
2. The locked Read is inserted into Locked Read reservation station 930A. The locked Write is inserted into Locked Write reservation station 932A.
3. When all previous reads in Read reservation station 930A have been completed, the locked Read is issued from Read reservation station 930A. Because all RmodW instructions are issued in order, the corresponding Write instruction is now also at the front of the Write reservation station 932A.
4. The Read now accesses scratchpad register 902. If it is locked, the Read waits until it is unlocked. Then it locks the scratchpad register 902 location, completes its operation and conveys the result to the result bus. However, the Read instruction is not yet removed from reservation station 930A.
5. All instructions dependent on the Read may now issue. Once the locked Write in Write reservation station 932A receives its input operand it is ready to issue. However, the Write does not issue until the Write is next to retire in the reorder buffer 32. This is required, because only non speculative writes are allowed.
6. When the Write is next to retire in the reorder buffer 32, the Write is issued from reservation station 932A and writes to scratchpad register 902. This write updates the data and unlocks the location. The SCINS Read and Write instructions are now removed from reservation stations 930 and 932, respectively.

In 6 above, the Read is removed from reservation station 930 if the associated Write retires. The Read must remain in the reservation station 930 in order to properly handle branch misprediction. In the case of a mispredicted branch, the mechanism operates as follows:

1. When a mispredicted branch is detected, the entries following the mispredicted branch in reorder buffer 32 are nullified.
2. During nullification, a SCINS locked instruction may be detected. This causes reorder buffer 32 to send the reorder buffer tags for the nullified SCINS to the SCINS functional unit together with an invalidate command. If multiple SCINS instructions are detected, then only the tags of the first instruction after the branch are required.
3. The SCINS functional unit uses the conveyed reorder buffer tag to invalidate all instructions in reservation stations 930 and 932 beginning with and following the received reorder buffer tag.
4. If the nullification of instructions in Read reservation station 930A hits a Read at the front of the reservation station 930A which has already been issued, the logic uses the address of

the Read instruction to unlock that location in the scratchpad register 902.

The mechanism described above allows a speculatively locked scratchpad location 902 to be unlocked. Advantageously, the above mechanism allows the speculative execution of locked reads. While the above discussion uses separate Read and Write reservation stations, other implementations may be used and are contemplated. However, it must be possible to unlock a location if a locked Read was mispredicted without executing the mispredicted locked Write.

Computer System

Turning now to Fig. 10, a block diagram of one embodiment of a computer system 1800 including multiprocessor computer 100 coupled to a variety of system components through a bus bridge 1802 is shown. Other embodiments are possible and contemplated. In the depicted system, a main memory 1804 is coupled to bus bridge 1802 through a memory bus 1806, and a graphics controller 1808 is coupled to bus bridge 1802 through an AGP bus 1810. Finally, a plurality of PCI devices 1812A-1812B are coupled to bus bridge 1802 through a PCI bus 1814. A secondary bus bridge 1816 may further be provided to accommodate an electrical interface to one or more EISA or ISA devices 1818 through an EISA/ISA bus 1820. Multiprocessor computer 100 is coupled to bus bridge 1802 through a CPU bus 1824 and to an optional L2 cache 1828.

Bus bridge 1802 provides an interface between multiprocessor computer 100, main memory 1804, graphics controller 1808, and devices attached to PCI bus 1814. When an operation is received from one of the devices connected to bus bridge 1802, bus bridge 1802 identifies the target of the operation (e.g. a particular device or, in the case of PCI bus 1814, that the target is on PCI bus 1814). Bus bridge 1802 routes the operation to the targeted device. Bus bridge 1802 generally translates an operation from the protocol used by the source device or bus to the protocol used by the target device or bus.

In addition to providing an interface to an ISA/EISA bus for PCI bus 1814, secondary bus bridge 1816 may further incorporate additional functionality, as desired. An input/output controller (not shown), either external from or integrated with secondary bus bridge 1816, may also be included within computer system 1800 to provide operational support for a keyboard and mouse 1822 and for various serial and parallel ports, as desired. An external cache unit (not shown) may further be coupled to CPU bus 1824 between multiprocessor computer 100 and bus bridge 1802 in other embodiments. Alternatively, the external cache may be coupled to bus bridge 1802 and cache control logic for the external cache may be integrated into bus bridge 1802. L2 cache 1828 is further shown in a backside configuration to processor 10. It is noted that L2 cache 1828 may be separate from multiprocessor computer 100, integrated into a cartridge (e.g. slot 1 or slot A) with multiprocessor computer 100, or even integrated onto a semiconductor substrate with multiprocessor computer 100.

Main memory 1804 is a memory in which application programs are stored and from which multiprocessor computer 100 primarily executes. A suitable main memory 1804 comprises DRAM (Dynamic Random Access Memory). For example, a plurality of banks of SDRAM (Synchronous DRAM) or Rambus DRAM (RDRAM) may be suitable.

PCI devices 1812A-1812B are illustrative of a variety of peripheral devices such as, for example, network interface cards, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards. Similarly, ISA device 1818 is illustrative of various types of peripheral devices, such as a modem, a sound card, and a variety of data acquisition cards such as GPIB or field bus interface cards.

Graphics controller 1808 is provided to control the rendering of text and images on a display 1826. Graphics controller 1808 may embody a typical graphics accelerator generally known in the art to render three-dimensional data structures which can be effectively shifted into and from main memory 1804. Graphics controller 1808 may therefore be a master of AGP bus 1810 in that it can request and receive access to a target interface within bus bridge 1802 to thereby obtain access to main memory 1804. A dedicated graphics bus accommodates rapid retrieval of data from main memory 1804. For certain operations, graphics controller 1808 may further be configured to generate PCI protocol transactions on AGP bus 1810. The AGP interface of bus bridge 1802 may thus include functionality to support both AGP protocol transactions as well as PCI protocol target and initiator transactions. Display 1826 is any electronic display upon which an image or text can be presented. A suitable display 1826 includes a cathode ray tube ("CRT"), a liquid crystal display ("LCD"), etc.

It is noted that, while the AGP, PCI, and ISA or EISA buses have been used as examples in the above description, any bus architectures may be substituted as desired. It is further noted that computer system 1800 may be a multiprocessing computer system including additional multiprocessor computers (e.g. multiprocessor computer 10a shown as an optional component of computer system 1800). Multiprocessor computer 10a may be similar to multiprocessor computer 10. More particularly, multiprocessor computer 10a may be an identical copy of multiprocessor computer 10. Multiprocessor computer 10a may be connected to bus bridge 1802 via an independent bus (as shown in Fig. 5) or may share CPU bus 1824 with processor 10. Furthermore, processor 10a may be coupled to an optional L2 cache 1828a similar to L2 cache 1828.

It is noted that the present discussion may refer to the assertion of various signals. As used herein, a signal is "asserted" if it conveys a value indicative of a particular condition. Conversely, a signal is "deasserted" if it conveys a value indicative of a lack of a particular condition. A signal may be defined to be asserted when it conveys a logical zero value or, conversely, when it conveys a logical one value. Additionally, various values have been described as being discarded in the above discussion. A value may be discarded in a number of manners, but generally involves modifying the value such that it is ignored by logic circuitry which receives the value. For example, if the value comprises a bit, the logic state of the value may be inverted to discard the value. If the value is an n-bit value, one of the n-bit encodings may indicate that the value is invalid. Setting the value to the invalid encoding causes the value to be discarded. Additionally, an n-bit value may include a valid bit indicative, when set, that the n-bit value is valid. Resetting the valid bit may comprise discarding the value. Other methods of discarding a value may be used as well.

Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

INDUSTRIAL APPLICABILITY

This invention is applicable to multi-processing computer systems.

WHAT IS CLAIMED IS:

1. A multiprocessor computer comprising:
   a plurality of processors, wherein said processors include a register file, a reorder buffer and circuitry to support speculative multithreaded execution; and
   a reorder buffer tag translation buffer; and
   a thread control device, wherein said thread control device is configured to store and transmit instructions between said processors, wherein said instructions support the speculative execution of threads on said processors, wherein said threads may be executed in parallel.

2. The multiprocessor computer of claim 1, wherein said circuitry is configured to support synchronization of said execution of said threads.

3. The multiprocessor computer recited in claim 1, wherein said translation buffer comprises a storage buffer, wherein said storage buffer maps an entry in a reorder buffer of a first processor to an entry in a reorder buffer of a second processor.

4. The multiprocessor recited in claim 1, wherein said thread control device is configured to synchronize execution of said threads on said processors.

5. The multiprocessor computer recited in claim 1, wherein a first processor of said plurality of processors is a master, wherein said master executes all single threaded code on said multiprocessor computer.

6. The multiprocessor computer of claim 5, further comprising a plurality of multithread instructions, wherein
   said master conveys data from a register file of said master to a second processor, in response to a first instruction of said plurality of multithread instructions; and
   said second processor conveys data from a register file of said second processor to said master, in response to a second instruction of said plurality of multithread instructions; and
   said second processor begins execution of a thread of instructions, in response to a third instruction of said plurality of multithread instructions; and
   said second processor ends execution of said thread of instructions, in response to a fourth instruction of said plurality of multithread instructions; and
   said master and said second processor synchronize thread execution, in response to a fifth instruction of said plurality of multithread instructions.

7. The multiprocessor computer of claim 6, wherein said first instruction, said second instruction and said third instruction are speculatively executed in said second processor.

8. A method comprising:

performing thread setup for execution of said thread on a second processor, wherein said setup comprises a first processor conveying setup instructions to said second processor, wherein said setup instructions are speculatively executed on said second processor; and

conveying a startup instruction from said first processor to said second processor, wherein said startup instruction is speculatively executed on said second processor; and

executing said thread on said second processor, in response to receiving said startup instruction, wherein said thread is speculatively executed, wherein said thread is executed in parallel with the execution of a thread on said first processor; and

terminating said execution of said thread on said second processor, wherein said termination comprises retiring a first termination instruction in said second processor; and

conveying results of said execution of said thread on said second processor to said first processor, in response to said second processor receiving a retrieve result instruction, wherein said retrieve result instruction is speculatively executed by said second processor.

9. The method recited in claim 8, further comprising:

storing a reorder buffer tag of a reorder buffer in said first processor in said first FIFO, wherein said reorder buffer tag corresponds to a location in said reorder buffer in said first processor containing said instruction; and

updating a reorder buffer tag translation buffer, wherein said update comprises storing in a location in said buffer a reorder buffer tag of said reorder buffer of said second processor, wherein said reorder buffer tag of said reorder buffer of said second processor corresponds to a location of said instruction in said reorder buffer of said second processor, wherein said location in said buffer is selected by said reorder buffer tag of said reorder buffer of said first processor.

10. The method recited in claim 8, further comprising synchronizing said execution of said thread on said first processor with said execution of said thread on said second processor.

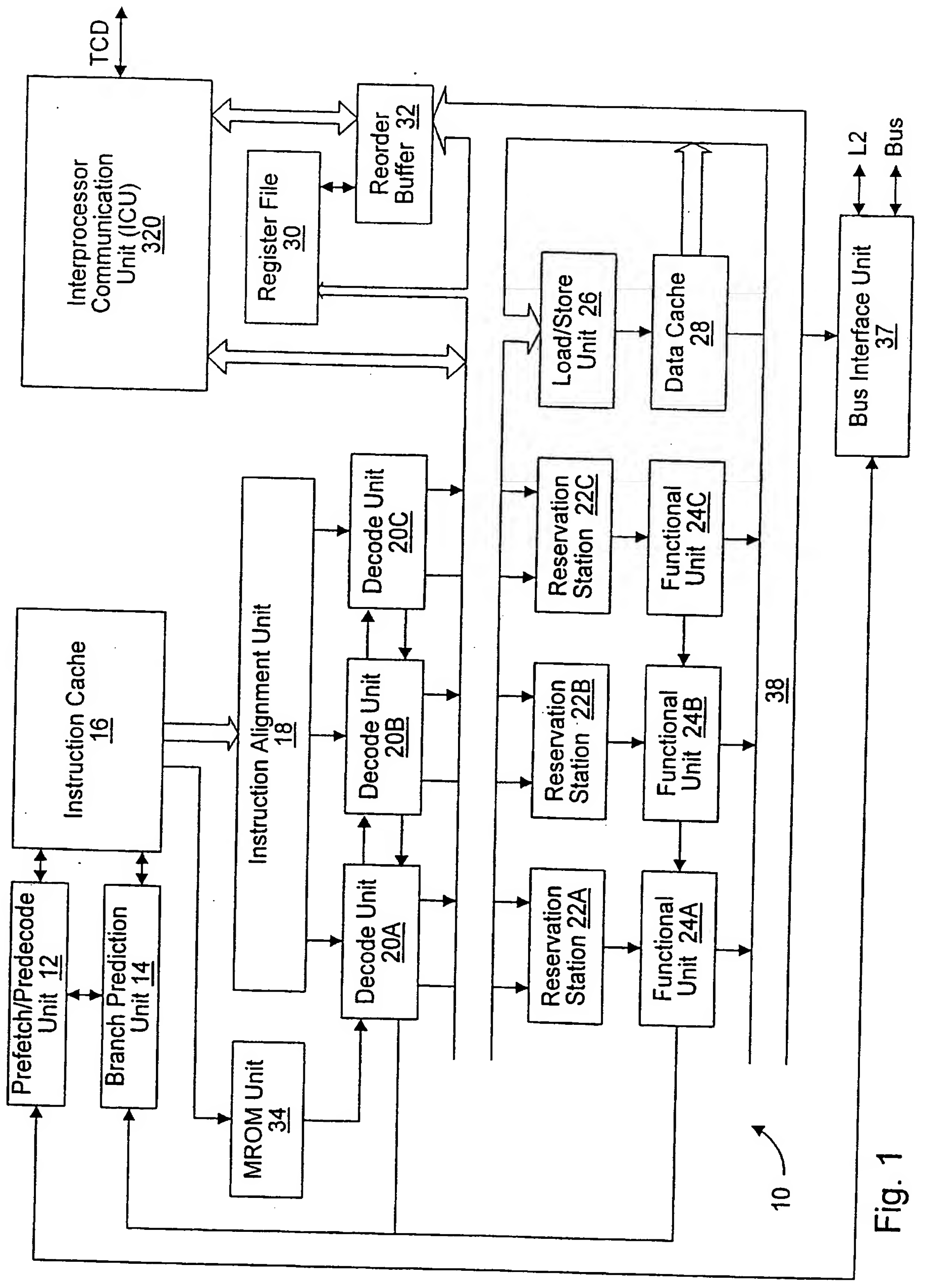

TCD
Interprocessor Communication Unit (ICU) 320
Reorder Buffer 32
Register File 30
Load/Store Unit 26
Data Cache 28
L2
Bus
Bus Interface Unit 37
Instruction Cache 16
Instruction Alignment Unit 18
Decode Unit 20C
Decode Unit 20B
Decode Unit 20A
Reservation Station 22C
Reservation Station 22B
Reservation Station 22A
Functional Unit 24C
Functional Unit 24B
Functional Unit 24A
38
Prefetch/Predecode Unit 12
Branch Prediction Unit 14
MROM Unit 34
10

Fig. 1

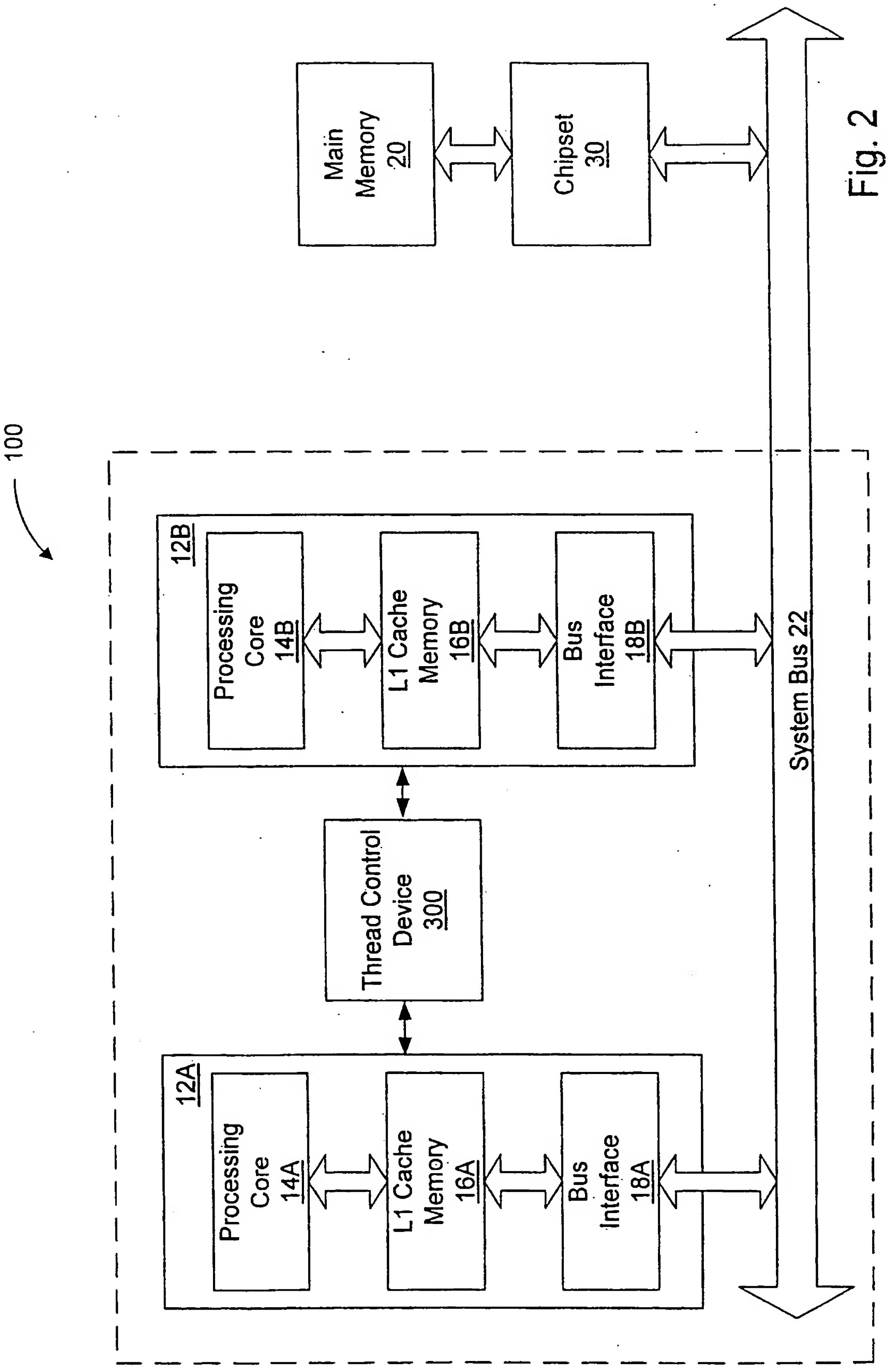

100
12B
Processing Core 14B
L1 Cache Memory 16B
Bus Interface 18B
Thread Control Device 300
12A
Processing Core 14A
L1 Cache Memory 16A
Bus Interface 18A
System Bus 22
Main Memory 20
Chipset 30

Fig. 2

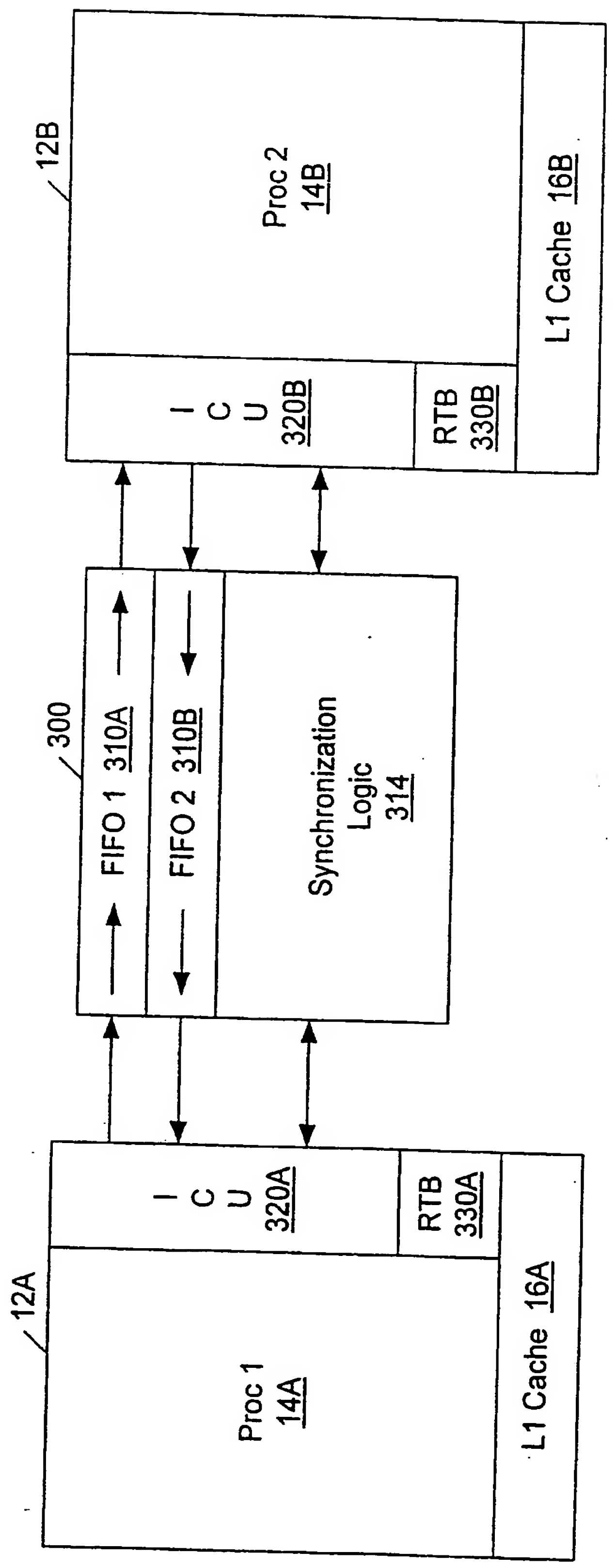

12B
Proc 2 14B
ICU 320B
RTB 330B
L1 Cache 16B
300
FIFO 1 310A
FIFO 2 310B
Synchronization Logic 314
12A
Proc 1 14A
ICU 320A
RTB 330A
L1 Cache 16A

Fig. 3A

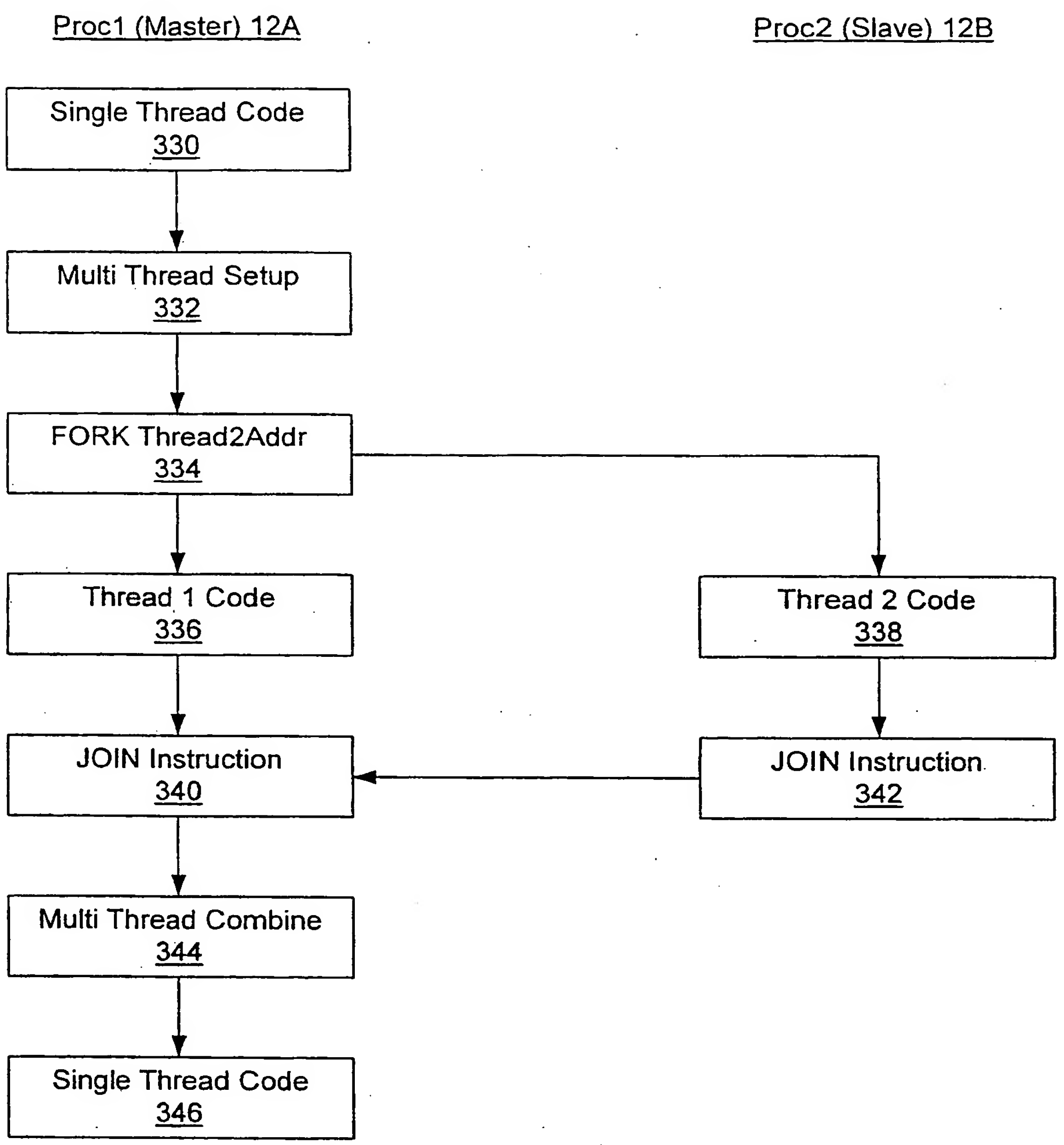

Proc1 (Master) 12A
Proc2 (Slave) 12B
Single Thread Code 330
Multi Thread Setup 332
FORK Thread2Addr 334
Thread 1 Code 336
Thread 2 Code 338
JOIN Instruction 340
JOIN Instruction 342
Multi Thread Combine 344
Single Thread Code 346

Fig. 3B

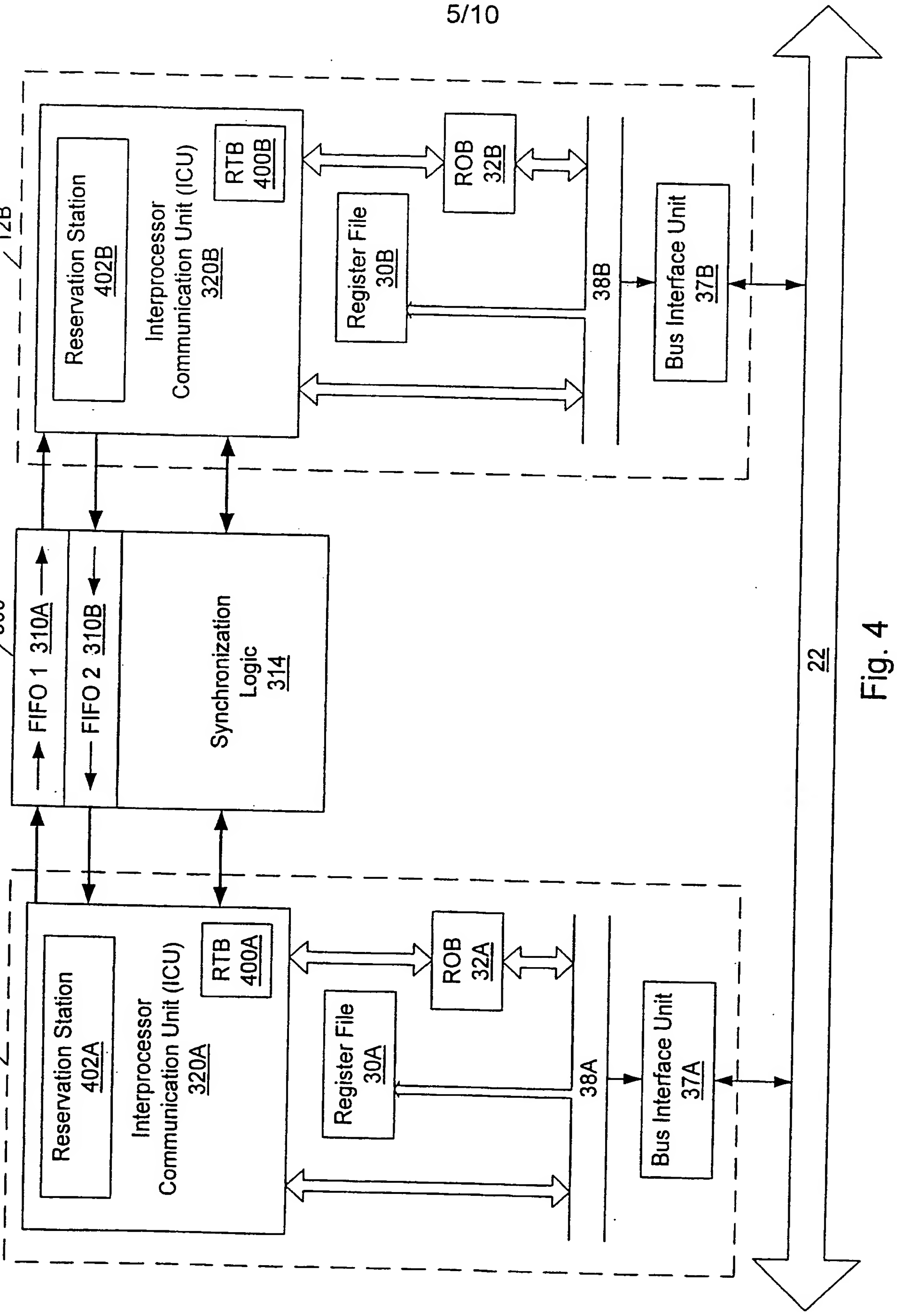

12B
Reservation Station
402B
Interprocessor
Communication Unit (ICU)
320B
RTB
400B
ROB
32B
Register File
30B
38B
Bus Interface Unit
37B
300
FIFO 1 310A
FIFO 2 310B
Synchronization
Logic
314
12A
Reservation Station
402A
Interprocessor
Communication Unit (ICU)
320A
RTB
400A
ROB
32A
Register File
30A
38A
Bus Interface Unit
37A
22

Fig. 4

	Thread 1 Code	Thread 2 Code	
500	L1:	mov reg1, Handle	510
502	Sync	Sync	512
504	RdFrProc 2, reg0, reg1	Sync	514
506	Sync		
507	cmp reg0, Handle		
508	jnz L1		

Fig. 5

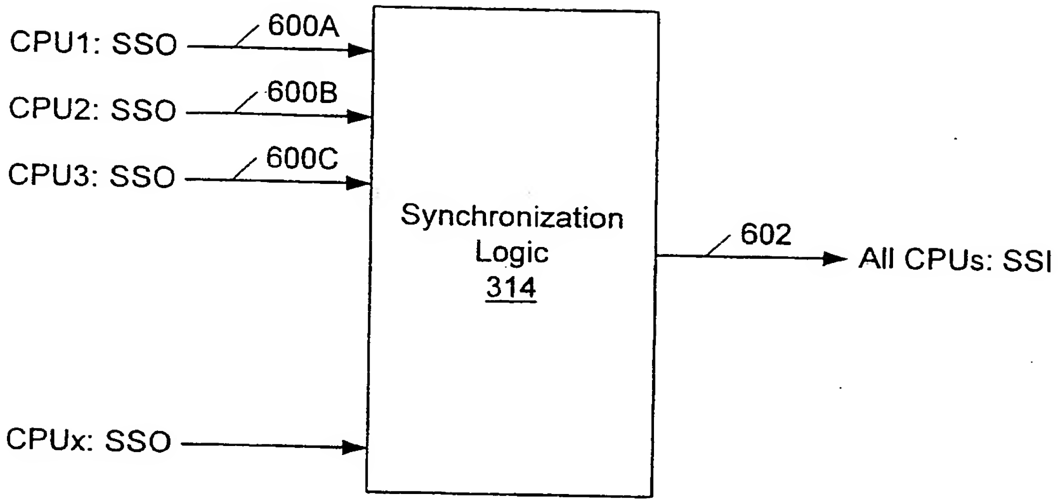

CPU1: SSO
600A
CPU2: SSO
600B
CPU3: SSO
600C
Synchronization Logic 314
602
All CPUs: SSI
CPUx: SSO

Fig. 6

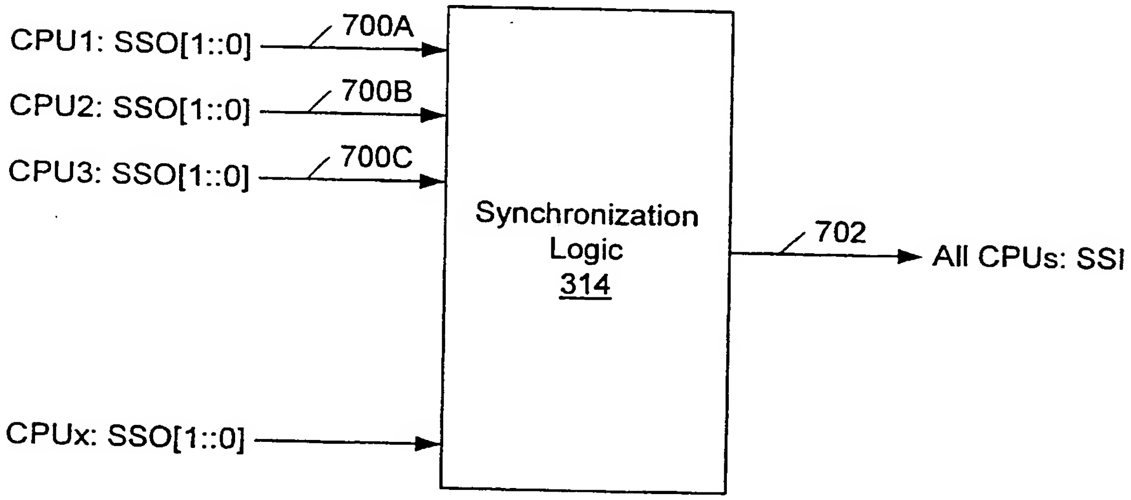

CPU1: SSO[1::0]
700A
CPU2: SSO[1::0]
700B
CPU3: SSO[1::0]
700C
Synchronization Logic 314
702
All CPUs: SSI
CPUx: SSO[1::0]

Fig. 7

```
WHILE (1)
{
    SWITCH (FSMState)
    {
        CASE: WAIT_ALL_S1
            SSI=0;
            IF (all SSO == S1)
                FSMState = ALL_S1;
            ELSE
                FSMState = WAIT_ALL_S1;

        CASE: ALL_S1
            FSMState = WAIT_ALL_S2;

        CASE: WAIT_ALL_S2
            SSI = 1;
            IF (all SSO == S2)
                FSMState = ALL_S2;
            ELSE
                FSMState = WAIT_ALL_S2;

        CASE: ALL_S2
            FSMState = WAIT_ALL_S1;
    }
}
```

Fig. 8

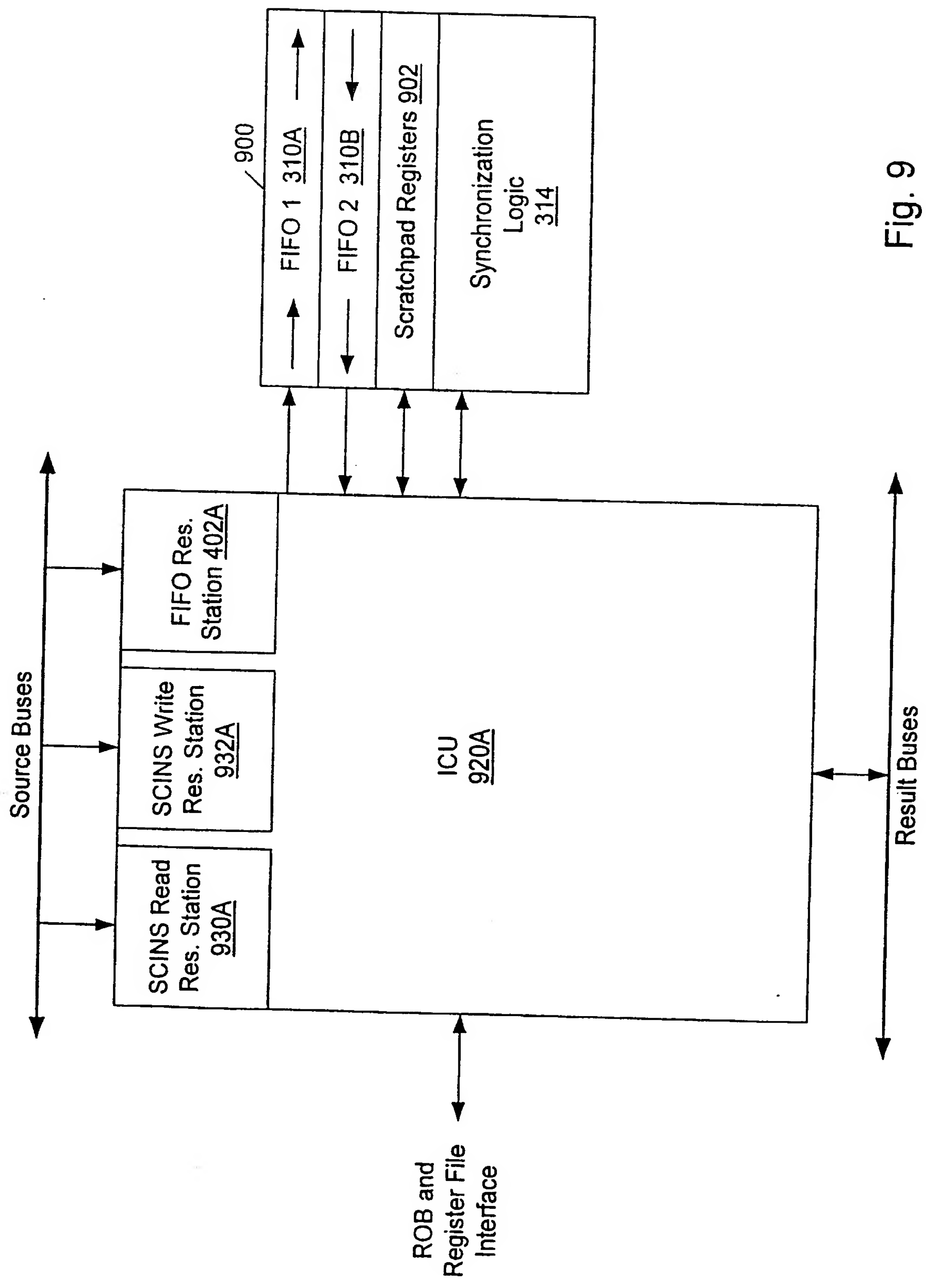

900
FIFO 1 310A
FIFO 2 310B
Scratchpad Registers 902
Synchronization Logic 314
Source Buses
FIFO Res. Station 402A
SCINS Write Res. Station 932A
SCINS Read Res. Station 930A
ICU 920A
Result Buses
ROB and Register File Interface

Fig. 9

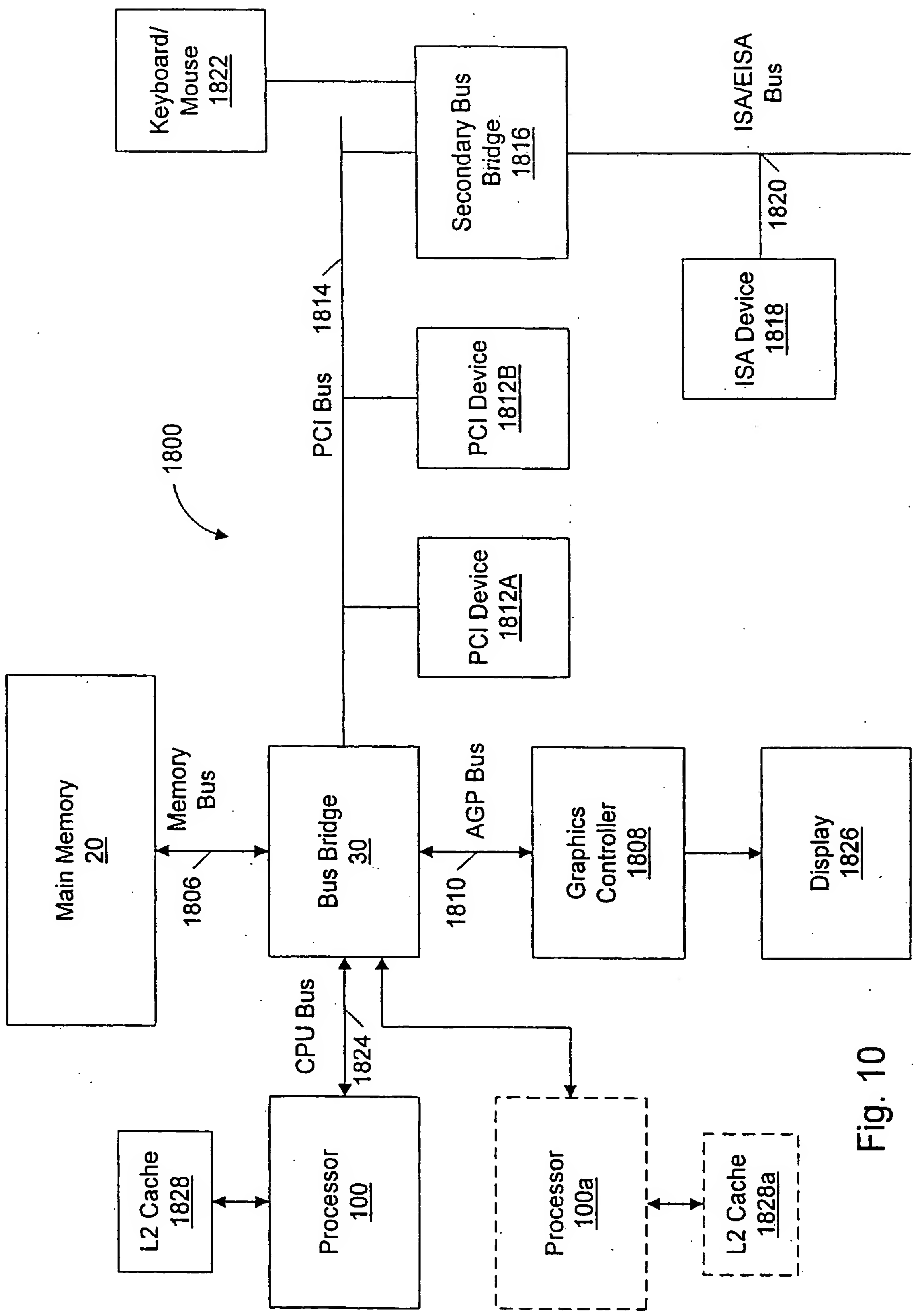

Keyboard/ Mouse 1822
Secondary Bus Bridge 1816
ISA/EISA Bus
1820
ISA Device 1818
1814
PCI Bus
PCI Device 1812B
1800
PCI Device 1812A
Main Memory 20
Memory Bus
1806
Bus Bridge 30
AGP Bus
1810
Graphics Controller 1808
Display 1826
CPU Bus
1824
L2 Cache 1828
Processor 100
Processor 100a
L2 Cache 1828a

Fig. 10

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 00/20881

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/48 G06F9/38

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, IBM-TDB, PAJ, INSPEC

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	HAMMOND L ET AL: "DATA SPECULATION SUPPORT FOR A CHIP MULTIPROCESSOR" ACM SIGPLAN NOTICES,ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK,US, vol. 33, no. 11, 1 November 1998 (1998-11-01), pages 58-69, XP000787299 ISSN: 0362-1340 the whole document --- -/--	1-10

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier document but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.
"&" document member of the same patent family

Date of the actual completion of the international search 26 March 2001	Date of mailing of the international search report 04/04/2001
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Michel, T

1

.national Application No
PCT/US 00/20881

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	MARCUELLO P ET AL: "SPECULATIVE MULTITHREADED PROCESSORS" MELBOURNE, AUSTRALIA, JULY 13 - 17, 1998,NEW YORK, NY: ACM,US, vol. 12TH CONF, 13 July 1998 (1998-07-13), pages 77-84, XP000895359 ISBN: 0-89791-998-X page 77, right-hand column, line 28 -page 80, left-hand column, line 10; figure 1 ---	1-10
A	KRISHNAN V ET AL: "A CHIP-MULTIPROCESSOR ARCHITECTURE WITH SPECULATIVE MULTITHREADING" IEEE TRANSACTIONS ON COMPUTERS,US,IEEE INC. NEW YORK, vol. 48, no. 9, September 1999 (1999-09), pages 866-880, XP000862501 ISSN: 0018-9340 the whole document ---	1,2,4
A	WO 98 43193 A (UNIV MARYLAND ;VISHKIN UZI (US)) 1 October 1998 (1998-10-01) the whole document ---	8-10
A	EP 0 725 335 A (IBM) 7 August 1996 (1996-08-07) the whole document ---	1,8
A	WO 93 02414 A (MASSACHUSETTS INST TECHNOLOGY) 4 February 1993 (1993-02-04) page 9, line 3 - line 20 page 13, line 24 -page 14, line 25; figure 1 -----	1,2,4-6, 8,10

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

rnational Application No
PCT/US 00/20881

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 9843193 A	01-10-1998	AU 6586898 A	20-10-1998
EP 0725335 A	07-08-1996	US 5724565 A	03-03-1998
		JP 3048910 B	05-06-2000
		JP 8249195 A	27-09-1996
WO 9302414 A	04-02-1993	US 5430850 A	04-07-1995
		EP 0596030 A	11-05-1994
		JP 7501163 T	02-02-1995
		US 5560029 A	24-09-1996